The Importance of Non-Data-Communication Overheads in MPI

P. Balaji[1], A. Chan[1], W. Gropp[2], R. Thakur[1] and E. Lusk[1]

[1]Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439, USA
{balaji, chan, thakur, lusk}@mcs.anl.gov

[2]Department of Computer Science,
University of Illinois, Urbana, IL, 61801, USA
wgropp@illinois.edu

Abstract

With processor speeds no longer doubling every 18-24 months owing to the exponential increase in power consumption and heat dissipation, modern HEC systems tend to rely lesser on the performance of single processing units. Instead, they rely on achieving high-performance by using the parallelism of a massive number of low-frequency/low-power processing cores. Using such low-frequency cores, however, puts a premium on end-host pre- and post-communication processing required within communication stacks, such as the message passing interface (MPI) implementation. Similarly, small amounts of serialization within the communication stack that were *acceptable* on small/medium systems can be brutal on massively parallel systems.

Thus, in this paper, we study the different non-data-communication overheads within the MPI implementation on the IBM Blue Gene/P system. Specifically, we analyze various aspects of MPI, including the MPI stack overhead itself, overhead of allocating and queueing requests, queue searches within the MPI stack, multi-request operations and various others. Our experiments, that scale up to 131,072 cores of the largest Blue Gene/P system in the world (80% of the total system size), reveal several insights into overheads in the MPI stack, which were previously not considered significant, but can have a substantial impact on such massive systems.

1 Introduction

Today's leadership class systems have already crossed the petaflop barrier. As we move further into the petaflop era, and look forward to multi petaflop and exaflop systems, we notice that modern high-end computing (HEC) systems are undergoing a drastic change in their fundamental architectural model. Due to the exponentially increasing power consumption and heat dissipation, processor speeds no longer double every 18-24 months. Accordingly, modern HEC systems tend to rely lesser on the performance of single processing units. Instead, they try to extract parallelism out of a massive number of low-frequency/low-power processing cores.

The IBM Blue Gene/L [4] was one of the early supercomputers to follow this architectural model, soon followed by other systems such as the IBM Blue Gene/P (BG/P) [11] and the SiCortex SC5832 [5]. Each of these systems uses processing cores that operate in a modest frequency range of 650–850 MHz. However, the capability of these systems is derived from the number of such processing elements they utilize. For example, the largest Blue Gene/L system today, installed at the Lawrence Livermore National Laboratory, comprises of 286720 cores. Similarly, the largest Blue Gene/P system, installed at the Argonne National Laboratory, comprises of 163840 cores.

While such an architecture provides the necessary ingredients for building petaflop and larger systems, the actual performance perceived by users heavily depends on the capabilities of the systems-software stack used, such

as the message passing interface (MPI) implementation. While the network itself is quite fast and scalable on these systems, the local pre- and post-data-communication processing required by the MPI stack might not be as fast, owing to the low-frequency processing cores. For example, local processing tasks within MPI that were considered *quick* on a 3.6 GHz Intel processor, might form a significant fraction of the overall MPI processing time on the modestly fast 850 MHz cores of a BG/P. Similarly, small amounts of serialization within the MPI stack which were considered *acceptable* on a system with a few hundred processors, can be brutal when running on massively parallel systems with hundreds of thousands of cores.

These issues raise the fundamental question on whether systems software stacks on such architectures would scale with system size, and if there are any fundamental limitations that researchers have to consider for future systems. Specifically, while previous studies have focused on communication overheads and the necessary improvements in these areas, there are several aspects that do not directly deal with data communication but are still very important on such architectures. Thus, in this paper, we study the non-data-communication overheads in MPI, using BG/P as a case-study platform, and perform experiments to identify such issues. We identify various bottleneck possibilities within the MPI stack, with respect to the slow pre- and post-data-communication processing as well as serialization points, and stress these overheads using different benchmarks. We further analyze the reasons behind such overheads and describe potential solutions for solving them.

The remaining part of the paper is organized as follows. We present a brief overview of the hardware and software stacks on the IBM BG/P system in Section 2. Detailed experimental evaluation examining various parts of the MPI stack are presented in Section 3. We present other literature related to this work in Section 4. Finally, we conclude the paper in Section 5.

2 BG/P Hardware and Software Stacks

In this section, we present a brief overview of the hardware and software stacks on BG/P.

2.1 Blue Gene/P Communication Hardware

BG/P has five different networks [12]. Two of them, 10-Gigabit Ethernet and 1-Gigabit Ethernet with JTAG interface[1], are used for File I/O and system management. The other three networks, described below, are used for MPI communication:

3-D Torus Network: This network is used for point-to-point MPI and multicast operations and connects all compute nodes to form a 3-D torus. Thus, each node has six nearest-neighbors. Each link provides a bandwidth of 425 MBps per direction for a total of 5.1 GBps bidirectional bandwidth per node.

Global Collective Network: This is a one-to-all network for compute and I/O nodes used for MPI collective communication and I/O services. Each node has three links to the collective network for a total of 5.1GBps bidirectional bandwidth.

Global Interrupt Network: It is an extremely low-latency network for global barriers and interrupts. For example, the global barrier latency of a 72K-node partition is approximately 1.3μs.

On the BG/P, compute cores do not handle packets on the torus network. A DMA engine on each compute node offloads most of the network packet injecting and receiving work, so this enables better overlap with of computation and communication. The DMA interfaces directly with the torus network. However, the cores handle the sending/receiving packets from the collective network.

[1] JTAG is the IEEE 1149.1 standard for system diagnosis and management.

2.2 Deep Computing Messaging Framework (DCMF)

BG/P is designed for multiple programming models. The Deep Computing Messaging Framework (DCMF) and the Component Collective Messaging Interface (CCMI) are used as general purpose libraries to support different programming models [17]. DCMF implements point-to-point and multisend protocols. The multisend protocol connects the abstract implementation of collective operations in CCMI to targeted communication networks. DCMF API provides three types of message-passing operations: two-sided send, multisend and one-sided get. All three have nonblocking semantics.

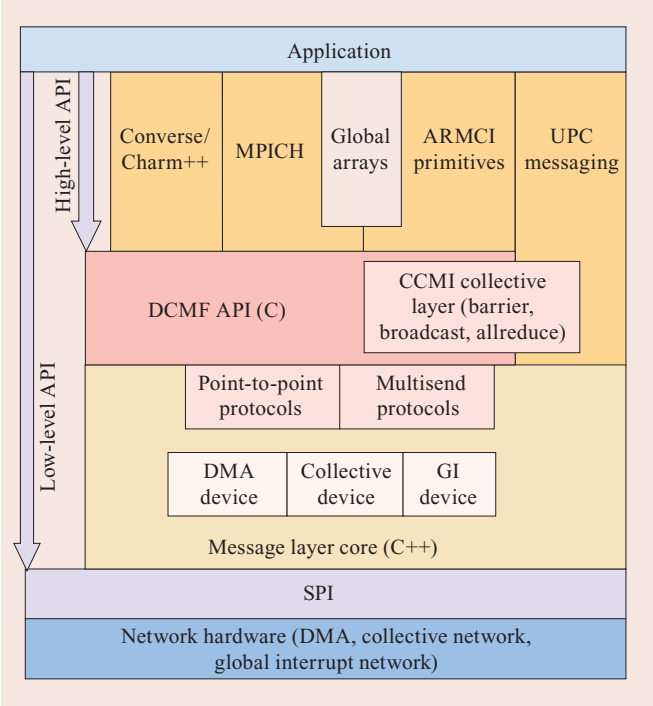


Figure 1: BlueGene/P Messaging Framework: CCMI=Component Collective Messaging Interface; DCMF=Deep Computing Messaging Framework; DMA=Direct Memory Access; GI=Global Interrupt; SPI=System Programming Interface. From IBM's BG/P overview paper [11].

2.3 MPI on DCMF

IBM's MPI on BG/P is based on MPICH2 and is implemented on top of DCMF. Specifically, the MPI on BG/P implementation borrows most of the upper-level code from MPICH2, including the ROMIO implementation of MPI-IO and MPE profiler, while implementing BG/P specific details within a device implementation called `dcmfd`. The DCMF library provides basic send/receive communication support. All advanced communication features such as allocation and handling of MPI requests, dealing with tags and unexpected messages, multi-request operations such as `MPI_Waitany` or `MPI_Waitall`, derived datatype processing and thread synchronization are **not** handled by the DCMF library and have to be taken care of by the MPI implementation.

3 Experiments and Analysis

In this section, we study the non-data-communication overheads in MPI on BG/P.

3.1 Basic MPI Stack Overhead

An MPI implementation can be no faster than the underlying communication system. On BG/P, this is DCMF. Our first measurements (Figure 2) compare the communication performance of MPI (on DCMF) with the communication performance of DCMF. For MPI, we used the OSU MPI suite [23] to evaluate the performance. For DCMF, we used our own benchmarks on top of the DCMF API, that imitate the OSU MPI suite. The latency test uses blocking communication operations while the bandwidth test uses non-blocking communication operations for maximum performance in each case.

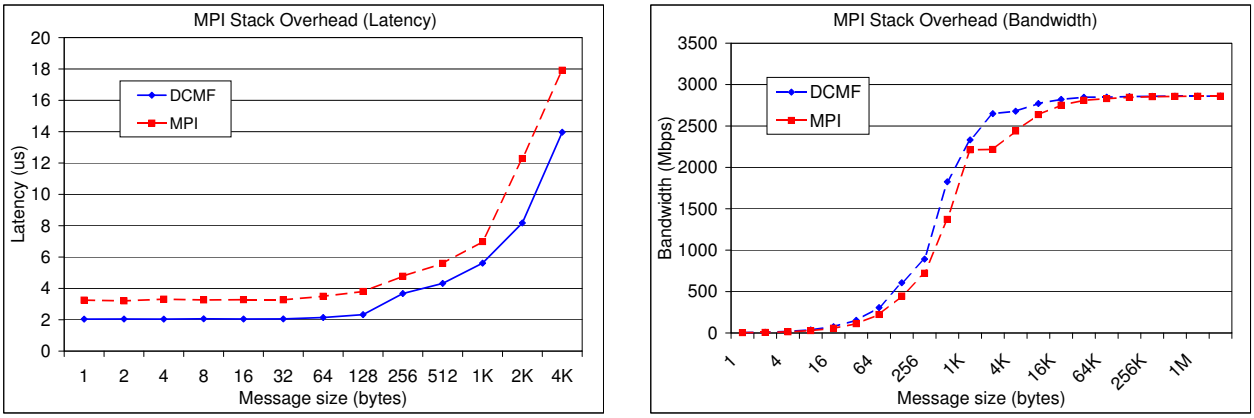


Figure 2: MPI stack overhead

The difference in performance of the two stacks is the overhead introduced by the MPI implementation on BG/P. We observe that the MPI stack adds close to 1.1μs overhead for small messages; that is, close to 1000 cycles are spent for pre- and post-data-communication processing within the MPI stack. We also notice that for message sizes larger than 1KB, this overhead is much higher (closer to 4μs or 3400 cycles). This additional overhead is because the MPI stack uses a protocol switch from eager to rendezvous for message sizes larger than 1200 bytes. Though DCMF itself performs the actual rendezvous-based data communication, the MPI stack performs additional book-keeping in this mode which causes this additional overhead. In several cases, such redundant book-keeping is unnecessary and can be avoided.

3.2 Request Allocation and Queueing Overhead

MPI provides non-blocking communication routines that enable concurrent computation and communication where the hardware can support it. However, from the MPI implementation's perspective, such routines require managing `MPI_Request` handles that are needed to wait on completion for each non-blocking operation. These requests have to be allocated, initialized and queued/dequeued within the MPI implementation for each send or receive operation, thus adding overhead, especially on low-frequency cores.

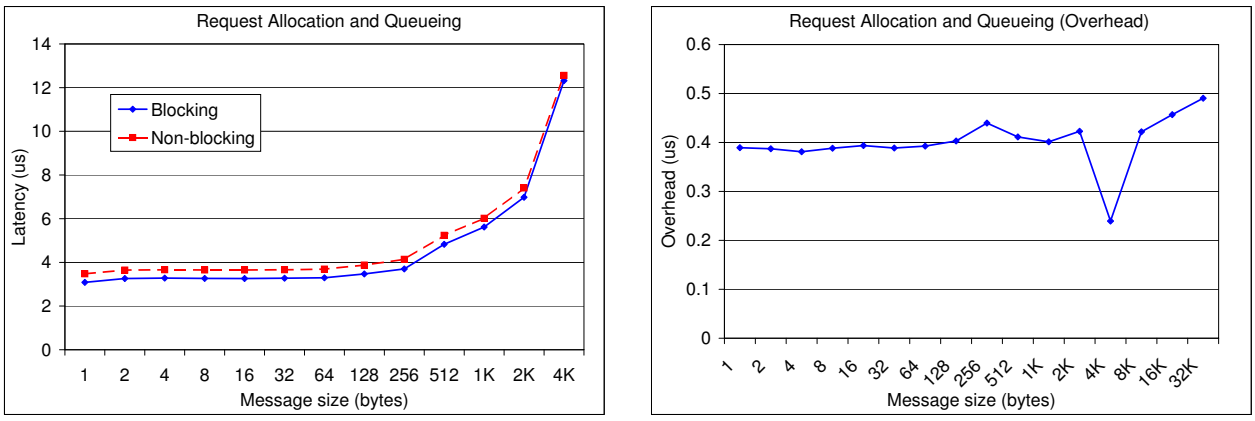


Figure 3: Request allocation and queuing: (a) Overall performance; (b) Overhead.

In this experiment, we measure this overhead by running two versions of the typical ping-pong latency test—one using `MPI_Send` and `MPI_Recv` and the other using `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall`. The latter incurs the overhead of allocating, initializing, and queuing/dequeuing request handles. Figure 3 shows that this overhead is roughly 0.4 μs or a little more than 300 clock cycles.[2] While this overhead is expected due to the number of request management operations, carefully redesigning them can potentially bring this down significantly.

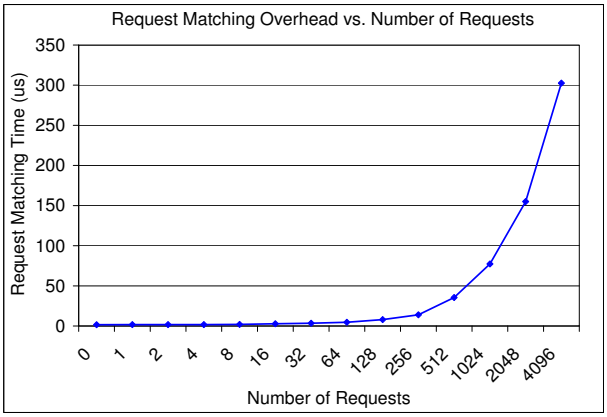
3.3 Overheads in Tag and Source Matching

MPI allows applications to classify different messages into different categories using tags. Each sent message carries a tag. Each receive request contains a tag and information about which source the message is expected from. When a message arrives, the receiver searches the queue of posted receive requests to find the one that matches the arrived message (both tag and source information) and places the incoming data in the buffer described by this request. Most current MPI implementations use a single queue for all receive requests, i.e., for all tags and all source ranks. This has a potential scalability problem when the length of this queue becomes large.

To demonstrate this problem, we designed an experiment that measures the overhead of receiving a message with increasing request-queue size. In this experiment, process P0 posts *M* receive requests for each of *N* peer processes with tag T0, and finally one request of tag T1 for P1. Once all the requests are posted (ensured through a low-level hardware barrier that does not use MPI), P1 sends a message with tag T1 to P0. P0 measures the time to receive this message not including the network communication time. That is, the time is only measured for the post-data-communication phase to receive the data after it has arrived in its local temporary buffer.

Figure 4 shows the time taken by the MPI stack to receive the data after it has arrived in the local buffer. Figures 4(a) and 4(b) show two different versions of the test—the first version keeps the number of peers to one (N = 1) but increases the number of requests per peer (M), while the second version keeps the number of requests per peer to one (M = 1) but increases the number of peers (N). For both versions, the time taken increases rapidly with increasing number of total requests ($M \times N$). In fact, for 4096 peers, which is modest considering the size BG/P can scale to, we notice that even just *one request per peer* can result in a queue parsing time of about

[2]This overhead is more than the entire point-to-point MPI-level shared-memory communication latency on typical commodity Intel/AMD processors [13].



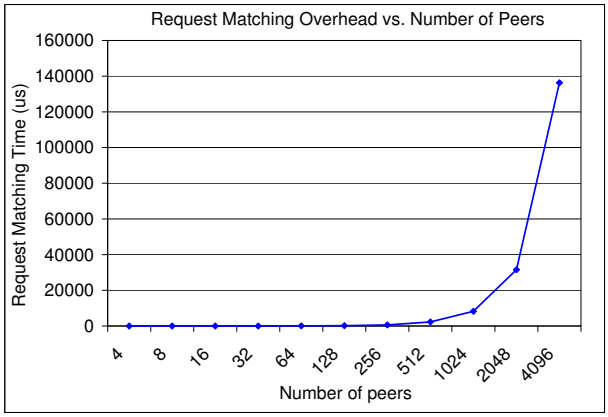


Figure 4: Request matching overhead: (a) requests-per-peer, (b) number of peers.

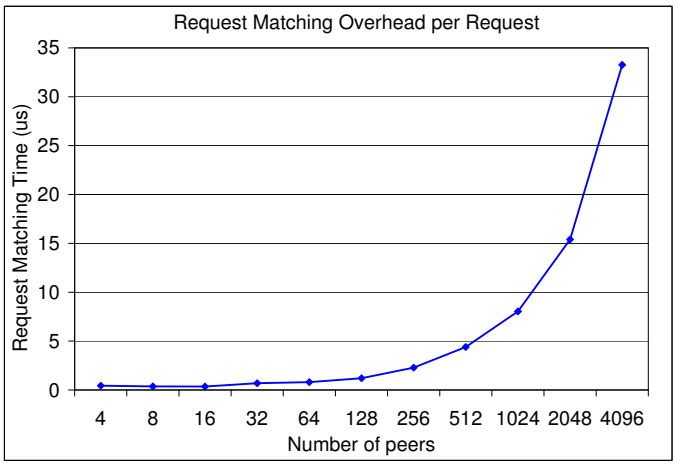


Figure 5: Matching overhead per request

140000μs.

Another interesting observation in the graph is that the time increase with the number of peers is not linear. To demonstrate this, we present the average time taken per request in Figure 5—the average time per request increases as the number of requests increases! Note that parsing through the request queue should take linear time; thus the time per request should be constant, not increase. There are several reasons for such a counter-intuitive behavior; we believe the primary cause for this is the limited number of pre-allocated requests that are reused during the life-time of the application. If there are too many pending requests, the MPI implementation runs out of these pre-allocated requests and more requests are allocated dynamically.

3.4 Algorithmic Complexity of Multi-request Operations

MPI provides operations such as `MPI_Waitany`, `MPI_Waitsome` and `MPI_Waitall` that allow the user to provide multiple requests at once and wait for the completion of one or more of them. In this experiment, we measure the MPI stack's capability to efficiently handle such requests. Specifically, the receiver posts several receive requests (`MPI_Irecv`) and once all the requests are posted (ensured through a low-level hardware barrier) the sender sends just one message that matches the first receive request. We measure the time taken to receive the message, not including the network communication time, and present it in Figure 6.

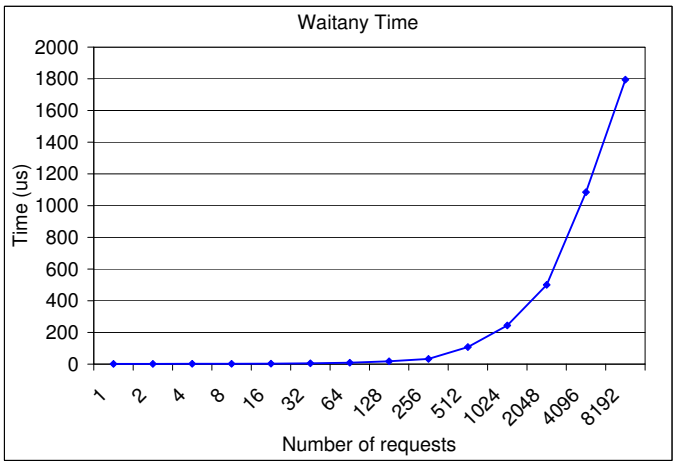


Figure 6: `MPI_Waitany` Time

We notice that the time taken by `MPI_Waitany` increases linearly with the number of requests passed to it. We expect this time to be constant since the incoming message matches the first request itself. The reason for this behavior is the algorithmic complexity of the `MPI_Waitany` implementation. While `MPI_Waitany` would have a worst-case complexity of `O(N)`, where `N` is the number of requests, its best-case complexity should be constant (when the first request is already complete when the call is made). However, the current implementation performs this in two steps. In the first step, it gathers the internal request handles for each request (takes `O(N)` time) and in the second step does the actual check for whether any of the requests have completed. Thus, overall, even in the best case, where the completion is constant time, acquiring of internal request handlers can increase the time taken linearly with the number of requests.

3.5 Overheads in Derived Datatype Processing

MPI allows non-contiguous messages to be sent and received using derived datatypes to describe the message. Implementing these efficiently can be challenging and has been a topic of significant research [16, 25, 8]. Depending on how densely the message buffers are aligned, most MPI implementations pack sparse datatypes into contiguous temporary buffers before performing the actual communication. This stresses both the processing power and the memory/cache bandwidth of the system. To explore the efficiency of derived datatype communication on BG/P, we looked only at the simple case of a single stride (vector) type with a stride of two. Thus, every other data item is skipped, but the total amount of data packed and communicated is kept uniform across the different datatypes (equal number of bytes). The results are shown in Figure 7.

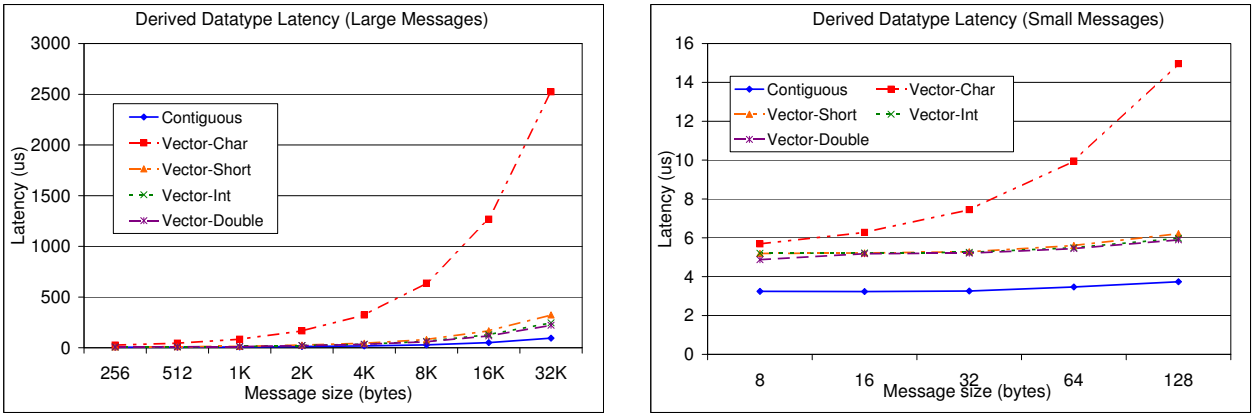


Figure 7: Derived datatype latency: (a) long messages and (b) short messages

These results show a significant gap in performance between sending a contiguous messages and a non-contiguous message (with the same number of bytes). The situation is particularly serious for a vector of individual bytes (`MPI_CHAR`). It is also interesting to look at the behavior for shorter messages (Figure 7(b)). This shows, roughly, a 2 μs gap in performance between contiguous send and a send of short, integer or double precision data with a stride of two.

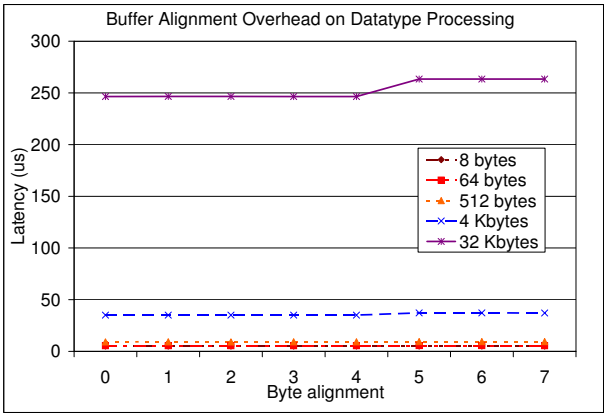
3.6 Buffer Alignment Overhead

For operations that involve touching the data that is being communicated (such as datatype packing), the alignment of the buffers that are being processed can play a role in overall performance if the hardware is optimized for specific buffer alignments (such as word or double-word alignments), which is common in most hardware today.

In this experiment (Figure 8), we measure the communication latency of a vector of integers (4 bytes) with a stride of 2 (that is, every alternate integer is packed and communicated). We perform the test for different alignment of these integers—"0" refers to perfect alignment to a double-word boundary, "1" refers to an misalignment of 1-byte. We notice that as long as the integers are within the same double-word (0-4 byte misalignment) the performance is better as compared to when the integers span two different double-words (5-7 byte misalignment), the performance difference being about 10%. This difference is expected as integers crossing the double-word boundary require both the double-words to be fetched before any operation can be performed on them.

3.7 Unexpected Message Overhead

MPI does not require any synchronization between the sender and receiver processes before the sender can send its data out. So, a sender can send multiple messages which are not immediately requested for by the receiver. When the receiver tries to receive the message it needs, all the previously sent messages are considered *unexpected*, and are queued within the MPI stack for later requests to handle. Consider the sender first sending multiple messages of tag T0 and finally one message of tag T1. If the receiver is first looking for the message with tag T1, it considers all the previous messages of tag T0 as *unexpected* and queues them in the unexpected



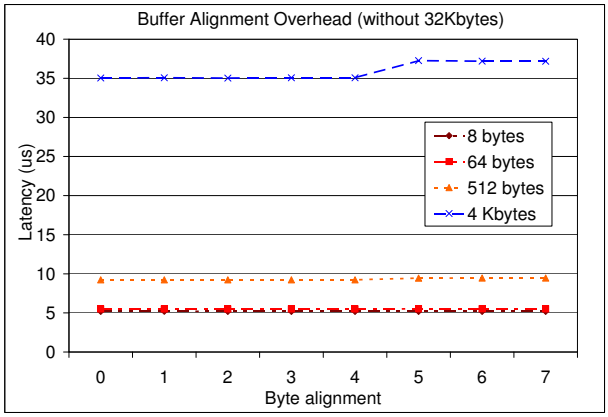
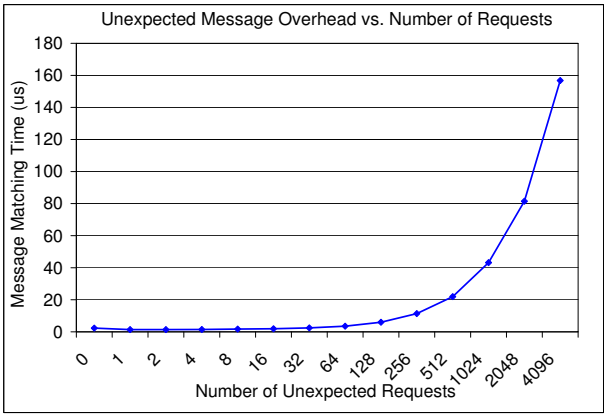


Figure 8: Buffer alignment overhead



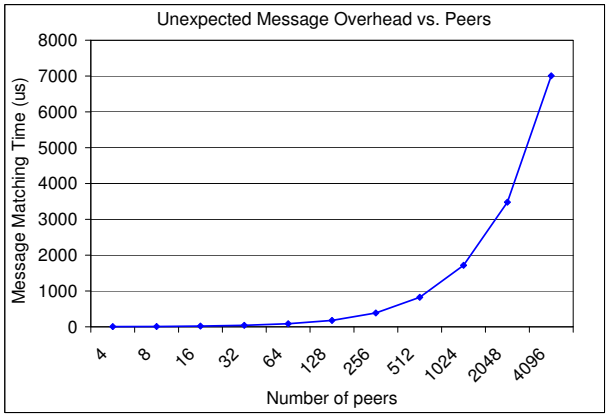


Figure 9: Unexpected message overhead: (a) Increasing number of messages per peer, with only one peer; (b) Increasing number of peers, with only one message per peer.

queue. Such queueing and dequeuing of requests (and potentially copying data corresponding to the requests) can add overhead.

To illustrate this, we designed an experiment that is a symmetric-opposite of the tag-matching test described in Section 3.3. Specifically, in the tag-matching test, we queue multiple receive requests and receive one message that matches the last queued request. In the unexpected message test, we receive multiple messages, but post only one receive request for the last received message. Specifically, process P0 first receives *M* messages of tag T0 from each of *N* peer processes and finally receives one extra message of tag T1 from P1. The time taken to receive the final message (tag T1) is measured, not including the network communication time, and shown in Figure 9 as two cases: (a) when there is only one peer, but the number of unexpected messages per peer increases (x-axis), and (b) the number of unexpected messages per peer is one, but the number of peers increases. We see that the time taken to receive the last message increases linearly with the number of unexpected messages.

3.8 Overhead of Thread Communication

To support flexible hybrid programming model such as OpenMP plus MPI, MPI allows applications to perform independent communication calls from each thread by requesting for `MPI_THREAD_MULTIPLE` level of thread concurrency from the MPI implementation. In this case, the MPI implementation has to perform appropriate locks within shared regions of the stack to protect conflicts caused due to concurrent communication by all threads. Obviously, such locking has two drawbacks: (i) they add overhead and (ii) they can serialize communication.

We performed two tests to measure the overhead and serialization caused by such locking. In the first test, we use four processes on the different cores which send 0-byte messages to `MPI_PROC_NULL` (these messages incur all the overhead of the MPI stack, except that they are never sent out over the network, thus imitating an infinitely fast network). In the second test, we use four threads with `MPI_THREAD_MULTIPLE` thread concurrency to send 0-byte messages to `MPI_PROC_NULL`. In the threads case, we expect the locks to add overheads and serialization, so the performance to be lesser than in the processes case.

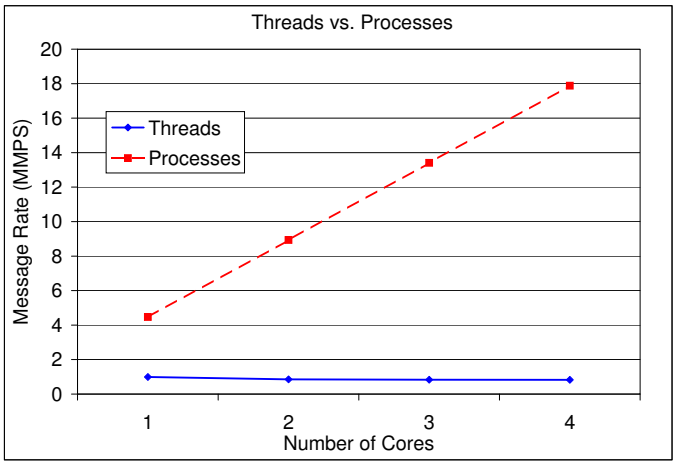


Figure 10: Threads vs. Processes

Figure 10 shows the performance of the two tests described above. The difference between the one-process and one-thread cases is that the one-thread case requests for the `MPI_THREAD_MULTIPLE` level of thread concurrency, while the one-process case requests for no concurrency, so there are no locks. As expected, in the process case, since there are no locks, we notice a linear increase in performance with increasing number of cores used. In the threads case, however, we observe two issues: (a) the performance of one thread is significantly lower than the performance of one process and (b) the performance of threads does not increase at all as we increase the number of cores used.

The first observation (difference in one-process and one-thread performance) points out the overhead in maintaining locks. Note that there is no contention on the locks in this case as there is only one thread accessing them. The second observation (constant performance with increasing cores) reflects the inefficiency in the concurrency model used by the MPI implementation. Specifically, most MPI implementations perform a global lock for each MPI operation thus allowing only one thread to perform communication at any given time. This results in virtu-

ally *zero* effective concurrency in the communication of the different threads. Addressing this issue is the subject of a separate paper [9].

3.9 Error Checking Overhead

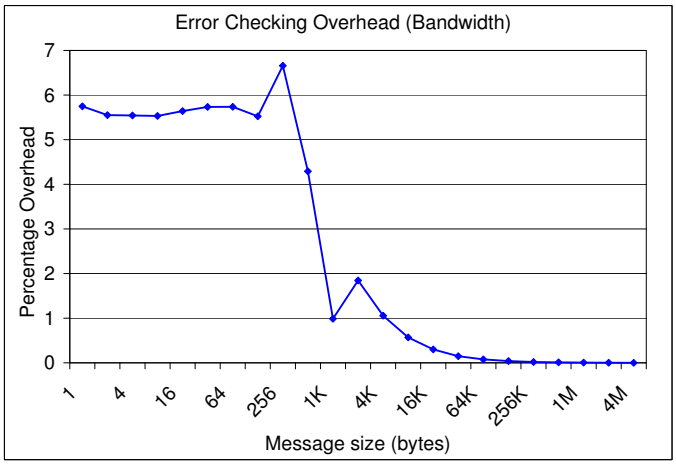


Figure 11: Error checking overhead

Since MPI is a library, it is impossible for the MPI implementation to check for user errors in arguments to the MPI routines except at runtime. These checks cost time; the more thorough the checking, the more time they take. MPI implementations derived from MPICH2 (such as the BG/P MPI) can be configured to enable or disable checking of user errors. Figure 11 shows the percentage overhead of enabling error checking. For short messages, it is about 5% of the total time, or around 0.1-0.2 μs. This overhead is relative small compared to the other overheads demonstrated in this paper, but should ideally be further reduced, by letting the user specify which parts of the code she would prefer to have error checking enabled, for example.

3.10 Non-Data Overheads in Sparse Vector Operations

A number of MPI collectives also have an associated vector version, such as `MPI_Gatherv`, `MPI_Scatterv`, `MPI_Alltoallv` and `MPI_Alltoallw`. These operations allow users to specify different data counts for different processes. For example, `MPI_Alltoallv` and `MPI_Alltoallw` allow applications to send a different amount of data to (and receive a different amount of data from) each peer process. This model is frequently used by applications to perform nearest neighbor kind of communication. Specifically, each process specifies 0 bytes for all processes in the communicator other than its neighbors.[3] The PETSc library [24], for example, uses `MPI_Alltoallw` in this manner.

For massive-scale systems such as Blue Gene/P, however, such communication would often result in a sparse data count array since the number of neighbors for each process is significantly smaller than the total number of

[3]We cannot perform such communication easily by using subcommunicators as each process would be a part of many subcommunicators, potentially causing deadlocks and/or serializing communication.

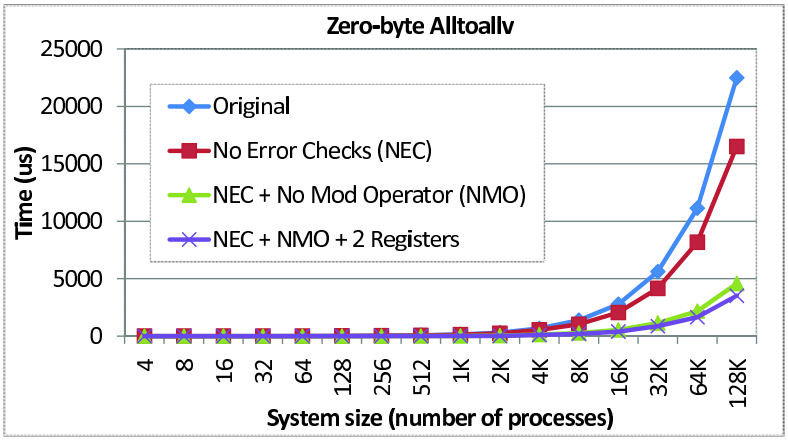


Figure 12: Zero-byte Alltoallv Communication

processes in the system. Thus, the MPI stack would spend a significant amount of time parsing the mostly empty array and finding the actual ranks of processes to which data needs to be sent to or received from. This overhead in illustrated in Figure 12 under the legend "original", where we measure the performance an extreme case of a sparse `MPI_Alltoallv` in which all data counts are zero. Performance numbers are shown for varying system sizes up to 131072 cores.

Together with the original scheme, we also present three different enhancements that allow can potentially allow the MPI library reduce this overhead, as described below.

No Error Checks (NEC): While useful for debugging and development, library-based runtime error checking, as described in Section 3.9, is generally an overhead for production runs of applications. Especially in the case of collective operations that take large arrays as input parameters, checking each array element for correctness is time consuming. Thus, we evaluated the performance of `MPI_Alltoallv` as described above (with zero data count) but after disabling error checks; this is illustrated under the legend "No Error Checks (NEC)". We notice that this enhancement gives about 25% benefit as compared to the base case (i.e., "Original").

No Mod Operator (NMO): In order to optimize the internal queue search time (as described in Sections 3.3 and 3.7), MPICH2 uses an offset-based loop to pick which destination to post a receive request from and a send request to. Specifically, this offset-based loop is implemented using a "%" arithmetic operator, so that a destination is picked as follows:

```
for i from 0 to communicator_size
      destination = (rank + i) % communicator_size
      if data_to_send(destination) != 0
            send_data(destination)
```

However, the "%" operator is an expensive operation on many architectures including x86 and PowerPC[4]. This is just an example of various operations that do not form a part of the first level optimizations, but can hamper performance on large-scale systems with moderately fast processors. That is, a few additional cycles per peer process would not cost too much on a fast processor or for a small number of peer processes. However, on systems such as BG/P, this can be a large performance bottleneck.

A simple approach to avoid using this operator, is to manually split the loop into two loops, one going from rank

[4]More details about this issue can be found here [6].

to the communicator_size, and the other going from zero to rank, as follows:

```
for i from rank to communicator_size
      if data_to_send(i) != 0
            send_data(destination)
for i from 0 to (rank - 1)
      if data_to_send(i) != 0
            send_data(destination)
```

This avoids the expensive "%" operator and improves performance significantly as illustrated in Figure 12.

Dual-register Loading: Many current processors allow for some vector-processor like operations allowing multiple registers to be loaded with a single instruction. The BG/P processor (PowerPC), in a similar manner, allows two 32-bit registers to be simultaneously loaded. Thus, since the data count array comprises of integers (which are 32-bit on BG/P), this allows us to compare two elements of the array to zero simultaneously. This can improve performance for sparse arrays without hurting performance for dense arrays. As illustrated in Figure 12, this benefit can be significant.

4 Related Work and Discussion

There has been significant previous work on understanding the performance of MPI on various architectures [20, 19, 18, 15, 10, 7]. However, all of this work mainly focuses on data communication overheads, which though related, is orthogonal to the study in this paper. Further, though not formally published, there are also proposals to extend MPI (in the context of MPI-3 [21]) to work around some of the overheads of existing collective operations such as `MPI_Alltoallv` for sparse communication.

Finally, there has also been a significant amount of work recently to understand whether MPI would scale to such massively large systems, or if alternative programming models are needed. This includes work in extending MPI itself [21] as well as other models including UPC [1], Co-Array Fortran [2], Global Arrays [3], OpenMP [22] and hybrid programming models (MPI + OpenMP [14], MPI + UPC). Some studies performed in this paper (such as request allocation and queueing overheads, buffer alignment overheads, overheads of multi-threading, and error checking) are independent of the programming model itself, and thus are relevant for other programming models too. However, some other studies in the paper (such as derived datatypes, `MPI_Alltoallv` communication) are more closely tied to MPI. For such parts, though they might not be directly relevant to other programming models, we believe that they do give an indication of potential pitfalls other models might run into as well.

5 Concluding Remarks

In this paper, we studied the non-data-communication overheads within MPI implementations and demonstrated their impact on the IBM Blue Gene/P system. We identified several bottlenecks in the MPI stack including request handling, tag matching and unexpected messages, multi-request operations (such as `MPI_Waitany`), derived-datatype processing, buffer alignment overheads and thread synchronization, that are aggravated by the low processing capabilities of the individual processing cores on the system as well as scalability issues triggered by the massive scale of the machine. Together with demonstrating and analyzing these issues, we also described potential solutions for solving these issues in future implementations.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-08ER25835.

References

[1] Berkeley Unified Parallel C (UPC) Project. http://upc.lbl.gov/.

[2] Co-Array Fortran. http://www.co-array.org/.

[3] Global Arrays. http://www.emsl.pnl.gov/docs/global/.

[4] http://www.research.ibm.com/journal/rd/492/gara.pdf.

[5] http://www.sicortex.com/products/sc5832.

[6] http://elliotth.blogspot.com/2007_07_01_archive.html, 2007.

[7] S. Alam, B. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. Vetter, P. Worley, and W. Yu. Early Evaluation of IBM BlueGene/P. In *SC*, 2008.

[8] P. Balaji, D. Buntinas, S. Balay, B. Smith, R. Thakur, and W. Gropp. Nonuniformly Communicating Noncontiguous Data: A Case Study with PETSc and MPI. In *IPDPS*, 2007.

[9] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *the Proceedings of the Euro PVM/MPI Users' Group Meeting*, Dublin, Ireland, 2008.

[10] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. Lusk. Non-Data-Communication Overheads in MPI: Analysis on Blue Gene/P. In *Euro PVM/MPI Users' Group Meeting*, Dublin, Ireland, 2008.

[11] Overview of the IBM Blue Gene/P project. http://www.research.ibm.com/journal/rd/521/team.pdf.

[12] IBM System Blue Gene Solution: Blue Gene/P Application Development. http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf.

[13] D. Buntinas, G. Mercier, and W. Gropp. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In *Euro PVM/MPI*, 2006.

[14] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, Washington, DC, USA, 2000. IEEE Computer Society.

[15] A. Chan, P. Balaji, R. Thakur, W. Gropp, and E. Lusk. Communication Analysis of Parallel 3D FFT for Flat Cartesian Meshes on Large Blue Gene Systems. In *HiPC*, Bangalore, India, 2008.

[16] W. Gropp, E. Lusk, and D. Swider. Improving the Performance of MPI Derived Datatypes. In *MPIDC*, 1999.

[17] S. Kumar, G. Dozsa, G. Almasi, D. Chen, M. Giampapa, P. Heidelberger, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *ICS*, 2008.

[18] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand Myrinet and Quadrics. In *Supercomputing 2003: The International Conference for High Performance Computing and Communications*, Nov. 2003.

[19] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[20] J. Liu, J. Wu, S. Kini, R. Noronha, P. Wyckoff, and D. K. Panda. MPI Over InfiniBand: Early Experiences. In *IPDPS*, 2002.

[21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, March 1994.

[22] Venkatesan Packirisamy and Harish Barathvajasankar. Openmp in multicore architectures. Technical report, University of Minnesota.

[23] D. K. Panda. OSU Micro-benchmark Suite. http://mvapich.cse.ohio-state.edu/benchmarks.

[24] PETSc library. `http://www.mcs.anl.gov/petsc`.

[25] R. Ross, N. Miller, and W. Gropp. Implementing Fast and Reusable Datatype Processing. In *Euro PVM/MPI*, 2003.